Loop and Data Transformations for Sparse Matrix Code

Anand Venkat, Mary Hall and Michelle Strout

Programming Language Design and Implementation, PLDI 2015
Motivation

• Sparse matrix applications
  • Iterative solvers for large-scale linear systems
  • Graph analytics

• **Problem:** Achieving high performance on sparse applications
  • Memory-bound
  • Irregular memory access patterns

• **Solution:** Specialize for input matrix structure/nonzero pattern
Motivation

- Manually tuned libraries provide high performance implementations
  - Reorganize matrix representation
  - Might modify data representation. e.g. pad with zeros

- **Disadvantages**
  - Must be manually ported as new architectures emerge
  - Difficult to compose optimizations

- **Problem Statement**: Can these optimizations be applied and composed within a compiler transformation framework for sparse matrix codes?
**Approach**

- **Input:** Sparse Computation
- **Inspector Code:** Matrix Format Conversion
- **Similar to library**
- **Executor Code:** Iterate using New Representation

---

- **CUDA-CHiLL**
  - **CHiLL**
    - Source-to-source Transformations
  - Polyhedral (Compile-time) Transformations
  - **Static/Affine Transformations**

- **Inspector/Executor (Run-time) Transformations**

- **Non-affine Transformations**

---

- **CHiLL/CUDA-CHiLL Transformation Recipe**
Sparse Matrix-Vector Multiply (SpMV)

for (i=0; i < n; i++)
  for (j=index[i]; j<index[i+1]; j++)
    y[i] += A[j] * x[col[j]];

Compressed Sparse Row (CSR) format

A =
\[
\begin{bmatrix}
1 & 5 & 0 & 0 \\
7 & 2 & 0 & 0 \\
0 & 0 & 3 & 6 \\
0 & 0 & 0 & 4
\end{bmatrix}
\]

A: [1 5 7 2 3 6 4]
index: [0 2 4 6 7]
col: [0 1 0 1 2 3 3]

Compressed Sparse Row (CSR)
Sparse Matrix Formats

Coordinate (COO)

\[ A: \begin{bmatrix} 1 & 5 & 0 & 0 \\ 7 & 2 & 0 & 0 \\ 0 & 0 & 3 & 6 \\ 0 & 0 & 0 & 4 \end{bmatrix} \]

DIA

\[ A: \begin{bmatrix} * & 7 & 0 & 0 \\ 1 & 2 & 3 & 4 \\ 5 & 0 & 6 & * \end{bmatrix} \]

offsets: [-1 0 1]

Block CSR (BCSR)

\[ \text{block-row-offset: [0 1 2]} \]

\[ \text{block-col: [0 2]} \]

Dense

ELL

\[ A: \begin{bmatrix} 1 & 5 \\ 7 & 2 \\ 3 & 6 \\ 4 & 0 \end{bmatrix} \]

\[ A: \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 2 & 3 \\ 3 & * \end{bmatrix} \]

Block CSR (BCSR)

\[ \text{block-row-offset: [0 1 2]} \]

\[ \text{block-col: [0 2]} \]
# Related Work: Compilers Supporting Sparse Codes

<table>
<thead>
<tr>
<th></th>
<th>Reordering Rows/Columns</th>
<th>Sparse Input Code</th>
<th>Composable with Polyhedral Transformations</th>
<th>Supports Fill-in</th>
<th>Automatic Matrix Format Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIPR (Shpeisman’99)</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bernoulli (Mateev’00)</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LL (Gilad et al.’10)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Sublimation/MT1 (Bik’96/van der Spek’11)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CHiLL/CUDA-CHiLL(‘15)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Contributions

• Transformations start off from *sparse* code

• Automatic generation of high performance Inspector/Executor Code

• Composition with other affine transformations within polyhedral model

• Demonstrate high-performance compiler-generated code on CPU and GPU
  • Performance compared to CUSP library on GPU
  • OSKI on CPU
Transformations

• **make-dense**: sparse $\rightarrow$ dense
  - Eliminates non-affine loop bounds
  - Introduces a guard for correctness
  - Enables further loop transformations such as tiling on dense loop

• **compact** and **compact-and-pad**: dense $\rightarrow$ sparse
  - Automatically generated inspector gathers iterations satisfying guard
  - Executor is the resulting transformed code
  - Executor can be further optimized by downstream polyhedral transformations

• **compact-and-pad** additionally performs a data transformation
  - Injecting zeros
  - Padding
Overview

Input Loop Nest

- guard
  - make-dense
    - Loop Nest with dense iterator(s)
      - Enabling Transformations
        - skew
        - tile
        - shift
        - ...
  - Data Transformation
    - Downstream Transformations
      - scalar expand
      - datacopy
      - unroll
      - ...
    - compact
    - Inspector Code
      - compact-and-pad
      - Executor Code
    - Output Code
**make-dense**

- **make-dense**: Exposes dense loop bounds

```c
for (i=0; i < n; i++)
    for (j=index[i]; j<index[i+1]; j++)
        y[i]+=a[j]*x[col[j]];
```

**make-dense(0,j)**

```c
for (i=0; i < n; i++)
    for(k=0; k < n; k++)
        for (j=index[i]; j<index[i+1]; j++)
            if(k==col[j])
                y[i]+=a[j]*x[k]
```

<table>
<thead>
<tr>
<th>j</th>
<th>col[j]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
compact

- **compact** derives a sparse iteration space from a dense one
- Executor code is in compacted CSR fashion

\[
\text{compact}(s_0, I_k)
\]

**Before**

\[
\begin{align*}
\text{for}(I_1) \\
\text{for}(I_{k-1}) \\
\text{for}(I_k) \\
\text{for}(I_{k+1}) \\
\text{for}(I_{k+n}) \\
\text{if}(\text{cond}(I_k)) \\
\text{s0: } \ldots X[I_k] \\
\end{align*}
\]

**After**

\[
\begin{align*}
\text{for}(I_1) \\
\text{for}(I_{k-1}) \\
\text{for}(I_k) = \text{offset_index}[I_1] \ldots [I_{k-1}]; I_k' < \text{offset_index}[I_1] \ldots [I_{k-1}+1]; I_k'++ \\
\text{for}(I_{k+1}) \\
\text{for}(I_{k+n}) \\
\text{if}(\text{cond}(\text{explicit_index}[I_k'])) \\
\text{s0: } \ldots X[\text{explicit_index}_{k'}[I_k']] \\
\end{align*}
\]
compact

• Generates inspector code
  • According to transformation specification
  • Constructs index arrays referenced by executor

• Executor can be subjected to downstream transformations

```
for(i=0; i < 2;i++)
    for(k=0; k < 5;k++)
        if(...)
            ... 
```
compact-and-pad

- **compact-and-pad** additionally copies the data footprint of the iterations satisfying the guard
- Pads with zero elements on selected iterations that do not satisfy the guard

\[
\text{compact-and-pad}(s0, I_k, A)
\]

Before

\[
\text{for}(I_1) \\
\quad \text{for}(I_{k-1}) \\
\quad \text{for}(I_k) \\
\quad \text{for}(I_{k+1}) \\
\quad \text{for}(I_{k+d}) \\
\quad \text{if}(\text{cond}(I_k)) \\
\quad \text{s0: } \ldots += \ldots \\
\quad A[\ldots]*X[I_k]\ldots
\]

Inner Loops

After

\[
\text{for}(I_1) \\
\quad \text{for}(I_{k-1}) \\
\quad \text{for}(I_k) \\
\quad \text{for}(I_{k+1}) \\
\quad \text{for}(I_{k+d}) \\
\quad \text{if}(\text{cond}(I_k)) \\
\quad \text{s0: } \ldots += \ldots A_{\text{prime}}[I_k][I_{k+1}][I_{k+d}] \\
\quad \text{A[\ldots]*X[explicit_index[I_k]]}\ldots
\]

Inner Loops
**compact-and-pad**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>∙</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>∙</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>∙</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pad with zeros**

**Eliminate entirely**

**compact-and-pad(k,...)**

**for(k=0; k < n;i++)**

**for(j=0; j < 5;j++)**

**if(...)**

**...**

**explicit_index(k')**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>k'</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**for(k=0; k < n;i++)**

**for(j=0; j < 5;j++)**

**if(...)**

**...**
Restrictions

• *make-dense*
  - Loop subjected to *make-dense* must not carry a dependence OR
  - Expression subjected to *make-dense* should be monotonically increasing OR
  - All dependences are due to an associative computation

• *compact*
  - Always legal

• *compact-and-pad*
  - Legal as long as the property of injectivity holds
Example : BCSR

After **make-dense**\((s0,j)\):

```cpp
for(i=0; i < n; i++)
  for(k=0; k < n; k++)
    for(j=index[i]; j < index[i+1]; j++)
      if(k==col[j])
        s0: y[i] += A[j]*x[k];
```

After tiling i and k loops by \(r\) and \(c\) respectively:

```cpp
for(ii=0; ii < n/r; ii++)
  for(kk=0; kk < n/c; kk++)
    for(i=0; i < r; i++)
      for(k=0; k < c; k++)
        for(j=index[ii*r+i]; j < index[ii*r+i+1]; j++)
          if(kk*c + k==col[j])
            s0: y[ii*r + i] += A[j]*x[kk*c +k];
```

After **compact-and-pad**\((s0, kk, A)\):

```cpp
for(ii=0; ii < n/r; ii++)
  for(kk=offset_index[ii]; kk < offset_index[ii+1]; kk++)
    for(i=0; i < r; i++)
      for(k=0; k < c; k++)
        for(j=index[ii*r+i]; j < index[ii*r+i+1]; j++)
          if(kk*c + k==col[j])
            s0: y[ii*r+i] += A_prime[kk][i][k]*x[explicit_index[kk]*c +k];
```
SpMV-BCSR CHiLL Script

```
make_dense(stmt,"j","k")

tile(stmt,"i", R, 1, counted)
tile(stmt,"k", C, 1, counted)

compact-and-pad(stmt,"kk","a","a_prime")

--downstream transformations
--copy to temporary storage and
--fully unroll inner loops

datacopy(executor_stmt,"k", x)
datacopy(executor_stmt, "i", y)
unroll(executor_stmt,"k", C)
unroll(executor_stmt, "i", R)
```
Experimental Methodology

- Codes generated using CUDA-CHiLL/CHiLL/CodeGen+/Omega+ polyhedral transformation and code generation framework
- DIA and ELL results on Nvidia Tesla K20c Kepler GPU
  - 13 Streaming Multiprocessors, 192 cores per SM.
  - 4.8 GB of global memory, 64KB register file per SM.
  - Benchmarked against CUSP
- BCSR was run on an Intel i7-4770(Haswell) CPU
  - 32GB DRAM, 8MB L3, 1MB L2 and 256KB L1 cache
  - Benchmarked against OSKI
- Matrices chosen were from the UFL Sparse Matrix Collection, Sam Williams’ group
Results

DIA Inspector Speedup

- Speedup over CUSP
- Matrices

ELL Inspector Speedup

- Speedup over CUSP
- Matrices

Performance of Compiler generated Inspectors and Executors competitive with CUSP

DIA Executor Performance

- Performance/GFLOPS
- Matrices

ELL Executor Performance

- Performance/GFLOPS
- Matrices
Results

**Inspector Code is 1.5x faster than OSKI**

**Executor Code within 1% of performance of OSKI**
Summary

• New loop and data transformations: *make-dense*, *compact* and *compact-and-pad*
  • Orchestrate the automatic generation of high performance Inspector/Executor Code
  • Allows for composition with other loop transformations within a polyhedral transformation and code generation framework

• Compiler generated inspectors and executors competitive with manually tuned libraries
  • Inspectors: Up to 1.5x faster and 1.27x for BCSR and DIA
  • Executors: Within 5% of CUSP and OSKI counterparts